Seminář Java

V

2005/2006

Radek Kočí

Rekapitulace

- Třídy: proměnné, metody, konstruktory, modifikátory přístupu, dědičnost
- Abstraktní třídy, vnořené třídy
- Hierarchie dědičnosti, porovnávání objektů (equals, hashCode)
- Rozhraní
- Datové typy: primitivní, objektové, pole
- Řídící konstrukce, operátory
- Ladění

Téma přednášky

- Výjimky
- Kontejnery: kolekce, seznamy

Výjimky

Co a k čemu jsou výjimky

- výjimka je mechanizmus umožňující psát robustní, spolehlivé programy
- robustní ve smyslu odolné proti chybám "okolí" – uživatele, systému, . . .
- výjimkami *v žádném případě* neošetřujeme chyby programu samotného! (hrubé zneužití)
- režie spojená s vyvoláním výjimky je vysoká!

Výjimky

Reprezentace výjimky

- výjimka (exception) je objekt třídy `java.lang.Exception`
- Objekty (výjimky) jsou vytvářeny (vyvolávány) buďto
  - automaticky běhovým systémem Javy, nastane-li nějaká běhová chyba, např. dělení nulou, nebo
  - jsou vytvořeny samotným programem, zdetekuje-li nějaký chybový stav, na nějž je třeba reagovat – např. do metody je předán špatný argument

Výjimky

Zpracování výjimky

- Vzniklý objekt výjimky je předán buďto:
  - v rámci metody, kde výjimka vznikla, do bloku `catch` $\Rightarrow$ výjimka je v bloku `catch` tzv. zachycena
  - výjimka "propadne" do nadřazené (volající) metody, kde je buďto v bloku `catch` zachycena nebo opět propadne atd.
- Výjimka tedy "putuje programem" tak dlouho, než je zachycena
- $\Rightarrow$ pokud není, běh JVM skončí s hlášením o výjimce

Syntaxe kódu s ošetřením výjimek

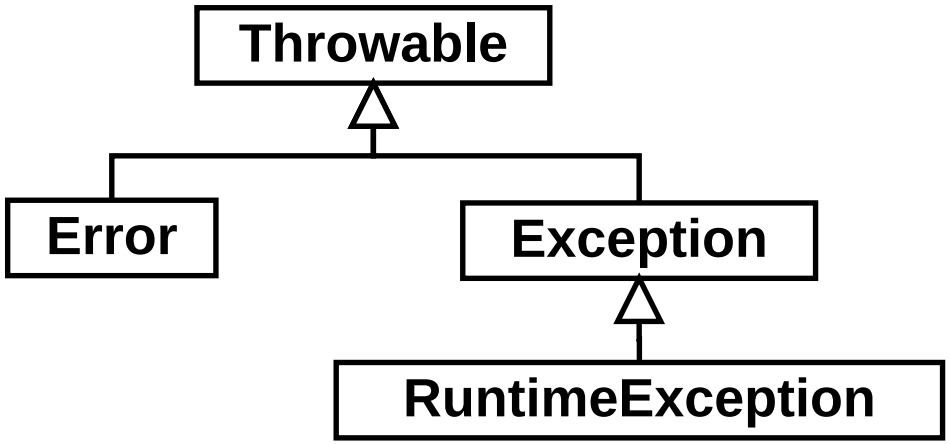
Základní syntaxe:

```
try {
      //zde může vzniknout výjimka
} catch (TypVýjimky proměnnáVýjimky) {
      // zde je výjimka ošetřena
      // je možné zde přistupovat k proměnnéVýjimky
}
```

- Bloku `try` se říká hlídaný blok, protože výjimky (příslušného hlídaného typu) zde vzniklé jsou zachyceny.
- V bloku `catch` jsou zachycené výjimky ošetřeny

Hierarchie výjimek

package `java.lang`



- `Throwable` – pouze objekty této třídy (a podtříd) mohou být generovány jako výjimky
- `Error` – vážné chyby JVM (*Out Of Memory, Stack Overflow, ...*)
- `Exception` – hlídané výjimky (checked exceptions)
- `RuntimeException` – běhové (runtime, nehlídané – unchecked) výjimky, takové výjimky nemusejí být zachytávány

Hlídané výjimky

Hlídané výjimky

- `java.lang.Exception`
- indikuje podmínky (stavy), které může aplikace chtít ošetřovat
- musí se explicitně uvádět
- hlídaná výjimka musí být zpracována!
- např. `java.io.FileNotFoundException`

Propuštění hlídané výjimky

- někdy není nutné či vhodné ošetřovat výjimku na místě
- metoda může deklarovat, že *propouští* výjimku (`throws`)
- klauzule `throws` říká, že během zpracování metody může být vygenerovaná uvedená výjimka, která není ošetřena
- volající metoda musí výjimku zpracovat (zachytit nebo propustit)

```
public FileReader(String fileName)
                          throws FileNotFoundException;

public void close() throws IOException;
```

Hlídané výjimky – ukázka

```
package seminar5;
import java.io.*;
public class OtevreniSouboru {
    public static void main(String[] args) {
        String jmeno = args[0];
        FileReader r;
        System.err.println("Otviram soubor "+jmeno);
        r = new FileReader(jmeno);
        System.err.println("Soubor otevren");
        r.close();
    }
}
```

```
unreported exception java.io.FileNotFoundException;
must be caught or declared to be thrown
         r = new FileReader(jmeno);
unreported exception java.io.IOException; must ...
         r.close();
```

Ošetření výjimek – hierarchie

- výjimku jsou hierarchicky uspořádané podle dědičnosti jejich tříd
- nadřazená výjimka pokrývá všechny odvozené výjimky
- př.: `FileNotFoundException` je speciálním případem `IOException`
- zpracováním `IOException` zpracujeme i `FileNotFoundException`

Zachycení výjimky

```
public static void main(String[] args) {
    String jmeno = args[0];
    FileReader r;
    System.err.println("Otviram soubor "+jmeno);
    try {
        r = new FileReader(jmeno);
        System.err.println("Soubor otevren");
        r.close();
    } catch (IOException ex) {
        System.err.println("Chyba pri manipulaci se
                                        souborem.");
    }
}
```

Zachycení výjimky

Je možné zpracovat každou výjimku zvlášť (*pozor na řazení podle hierarchie!*)

```
public static void main(String[] args) {
    String jmeno = args[0];
    FileReader r;
    System.err.println("Otviram soubor "+jmeno);
    try {
        r = new FileReader(jmeno);
        System.err.println("Soubor otevren");
        r.close();
    } catch (FileNotFoundException ex) {
        System.err.println("Soubor nelze otevrit.");
    } catch (IOException ex) {
        System.err.println("Chyba pri manipulaci se
                                        souborem.");
    }
}
```

Zachycení výjimky – chyba

```
public static void main(String[] args) {
    String jmeno = args[0];
    FileReader r;
    System.err.println("Otviram soubor "+jmeno);
    try {
        r = new FileReader(jmeno);
        System.err.println("Soubor otevren");
        r.close();
    } catch (IOException ex) {
        System.err.println("Chyba pri manipulaci se
                                    souborem.");
    }
    // Následující blok se nikdy neprovede !!!
    catch (FileNotFoundException ex) {
        System.err.println("Soubor nelze otevrit.");
    }
}
```

Propuštění výjimky

```
public FileReader(String fileName)
                            throws FileNotFoundException;
public void close() throws IOException;
```

```
public class OtevreniSouboru {
    static void otevri(String jmeno) {
        System.err.println("Otviram soubor "+jmeno);
        FileReader r = new FileReader(jmeno);
        r.close();
    }
    public static void main(String[] args) {
        otevri(args[0]);
        System.err.println("Soubor otevren");
    }
}
```

Propuštění výjimky

```
public class OtevreniSouboru {
    static void otevri(String jmeno) throws IOException
    {
        System.err.println("Otviram soubor "+jmeno);
        FileReader r = new FileReader(jmeno);
        r.close();
    }

    public static void main(String[] args) {
        try {
            otevri(args[0]);
            System.err.println("Soubor otevren");
        } catch (IOException ioe) {
            System.err.println("Nelze otevrit soubor");
        }
    }
}
```

Nehlídané výjimky

Nehlídané výjimky

- `java.lang.RuntimeException`
- výjimky, které mohou být generovány během standardních operací JVM
- nemusí se explicitně uvádět
- nemusí se zachytávat
- např. `java.lang.ArrayIndexOutOfBoundsException`, `java.lang.NullPointerException`

Generování výjimky

- výjimku lze generovat (klíčové slovo `throw`)

```
public abstract class Reader ...
{
  public void mark(int limit) throws IOException
  {
    throw new IOException("mark() not supported");
  }
}
```

```
public abstract class Reader ...
{
  public InputStreamReader(..., String charsetName) ...
   {
    if (charsetName == null)
      throw new NullPointerException("charsetName");
  }
}
```

Klauzule `finally`

Klauzule (blok) `finally`:

- může následovat ihned po bloku `try` nebo až po blocích `catch`
- slouží k "úklidu v každém případě", tj.
  - když je výjimka zachycena blokem `catch`
  - i když je vygenerovaná jiná než ošetřovaná výjimka
  - i když je výjimka propuštěna do volající metody
- Používá se typicky pro uvolnění systémových zdrojů – uzavření souborů ...

Vlastní výjimky

- typy (tj. třídy) výjimek si můžeme definovat sami
- bývá zvykem končit názvy tříd (výjimek) na `Exception`
- *je lepší využívat standardní výjimky!*

```
class MyException extends Exception {
     protected int pocetParametru;

     public MyException(int pocet) {
          pocetParametru = pocet;
     }

     public int getPocetParametru() {
          return pocetParametru;
     }
}
```

Ukázka vlastní výjimky a klauzule finally

```
public static void main(String[] args) {
   int pocetParametru = args.length;
   try {
      if (pocetParametru < 2)
          throw new MyException(pocetParametru);
      System.out.println("Spravny pocet parametru: "
                         + pocetParametru);

   } catch (MyException mp) {
      System.out.println("Malo parametru:  "
                        + mp.getPocetParametru());
   } finally {
      System.out.println("Konec");
   }
}
```

Reakce na výjimku

Jak můžeme reagovat?

- Napravit příčiny vzniku chybového stavu – např. znovu nechat načíst vstup
- Poskytnout za chybný vstup náhradu – např. implicitní hodnotu
- Operaci neprovést ("vzdát") a sdělit chybu výše tím, že výjimku "propustíme" z metody

Výjimková pravidla:

- Vždy nějak reagujme! Neignorujme, nepotlačujme, tj.
- blok `catch` nenechávejme prázdný, přinejmenším vypišme `e.printStackTrace()`
- Nelze-li reagovat na místě, propusťme výjimku výše (a popišme to v dokumentaci...)

Ukázka nesprávného použití výjimky

```
public static void main(String[] args) {
    try {
        otevri(args[0]);
    }
    catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Neni zadan argument.");
    }
}
```

```
public static void main(String[] args) {
    if (args.length == 0) {
         System.err.println("Neni zadan argument.");
         return;
    }
    otevri(args[0]);
}
```

Ukázka nesprávného použití výjimky

```
try {
    int i = 0;
    while(true) {
        a[i++] = i;
    }
} catch (ArrayIndexOutOfBoundsException e) {}
```

```
int length = a.length;
for(int i = 0; i < length; ) {
    a[i++] = i;
}
```

Kontejnery

Kontejnery (containers) v Javě

- slouží k ukládání objektů (ne hodnot primitivních typů!)
- verze < 5.0
  - kontejnery jsou koncipovány jako beztypové
- verze => 5.0
  - kontejnery nesou typovou informaci o prvcích

Většinou se používají kontejnery hotové, vestavěné (součást Java Core API):

- vestavěné kontejnerové třídy jsou definovány v balíku `java.util`
- je možné vytvořit si vlastní implementace, obvykle ale zachovávající/implementující "standardní" rozhraní

Kontejnery

K čemu slouží?

- jsou dynamickými alternativami k poli a mají daleko širší použití
- k uchování proměnného počtu objektů
- počet prvků se v průběhu existence kontejneru může měnit
- oproti polím nabízejí časově efektivnější algoritmy přístupu k prvkům

Kontejnery

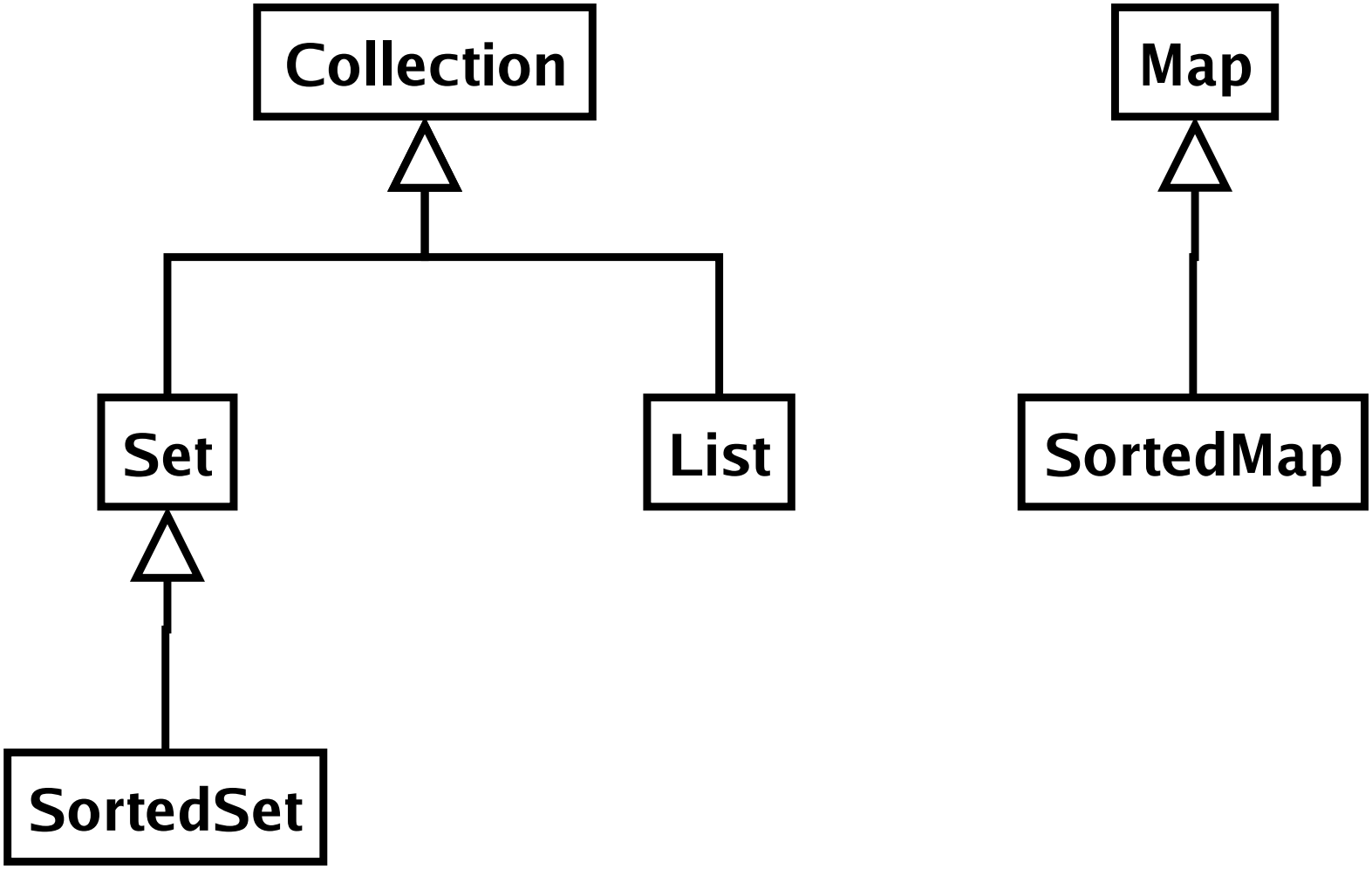
Základní kategorie kontejnerů

- seznam (`List`) – lineární struktura
- množina (`Set`) – struktura bez uspořádání, rychlé dotazování na přítomnost prvku
- asociativní pole, mapa (`Map`) – struktura uchovávající dvojice `klíč`→`hodnota`, rychlý přístup přes klíč

Kontejnery – rozhraní, nepovinné metody

- Funkcionalita vestavěných kontejnerů je obvykle předepsána výhradně rozhraním, které implementují.
- Rozhraní však připouštějí, že některé metody jsou nepovinné, třídy jej nemusí implementovat (*optional operations*)!
- V praxi se totiž někdy nehodí implementovat jak čtecí, tak i zápisové operace – některé kontejnery jsou "read-only"

Kontejnery – rozhraní



Kontejnery – implementace rozhraní

	hash table	*resizable array*	*balanced tree*	*linked list*
`Set`	`HashSet`		`TreeSet`	
`List`		`ArrayList`		`LinkedList`
`Map`	`HashMap`		`TreeMap`	

`SortedSet`
`SortedMap`

Uvedené kontejnery

- implementují všechny *optional* operace
- povolují `null` elementy (klíče, hodnoty)
- jsou nesynchronizované

Kontejnery – souběžný přístup, výjimky

- Moderní kontejnery jsou nesynchronizované, nepřipouštějí souběžný přístup z více vláken.
- Standardní, nesynchronizovaný, kontejner lze však "zabalit" synchronizovanou obálkou.
- Při práci s kontejnery může vzniknout řada výjimek, např. `IllegalStateException` apod.
- Většina má charakter výjimek běhových, není povinností je odchytávat – pokud věříme, že nevzniknou.

Kolekce

Kolekce

- jsou kontejnery implementující rozhraní `Collection` (viz *API doc k rozhr. Collection*)
- Rozhraní kolekce popisuje velmi obecný kontejner, disponující operacemi: přidávání, rušení prvku, získání iterátoru, zjišťování prázdnosti atd.
- Mezi kolekce patří mimo `Map` všechny ostatní vestavěné kontejnery – `List`, `Set`
- Prvky kolekce nemusí mít svou pozici danou indexem – viz např. `Set`

Kontejnery – Java 1.4.2

```
public interface List extends Collection {
  ...
  public boolean add(Object o);
  public Object get(int index);
}
```

- při vybírání elementu z kolekce musíme *přetypovat*
- není hlídáno kompilátorem
- pouze dynamická typová kontrola
- $\Rightarrow$ šance na vygenerování run-time výjimky

Kontejnery – Java 5.0

Generics (vlastnost Java 5.0)

```
public interface List<E> extends Collection<E> {
  ...
  public boolean add(E o);
  public E get(int index);
}
```

- umožňuje komunikaci s kompilátorem
- statická kontrola typů při manipulaci s kontejnery
- bez přetypování!

Iterátory

Iterátory jsou prostředkem, jak "chodit" po prvcích kolekce buď to

- v neurčeném pořadí nebo
- v uspořádání (u uspořádaných kolekcí)

Každý iterátor musí implementovat velmi jednoduché rozhraní `Iterator` se třemi metodami:

- `boolean hasNext()`
- `Object next()` (resp. `E next()`)
- `void remove()`

Seznamy

Seznamy

- lineární struktury
- implementují rozhraní `List`
- prvky lze adresovat indexem (typu `int`)
- poskytují možnost získat dopředný i zpětný iterátor
- lze pracovat i s podseznamy

Standardní implementace

- `ArrayList`
- `LinkedList`

Seznamy – příklad

```
class Clovek {
    String name;

    public Clovek(String n) {
        name = n;
    }

    public void vypisInfo() {
        System.out.println("Clovek jmenem " + name);
    }
}
```

Seznamy – příklad (1.4.2)

Vytvoříme seznam, naplníme jej a chodíme po položkách iterátorem.

```
public static void main(String[] args) {
    List seznam = new ArrayList();
    seznam.add(new Clovek("Ferda"));
    seznam.add(new Clovek("Franta"));
    seznam.add(new Clovek("Ferenc"));

    for (Iterator i = seznam.iterator(); i.hasNext();)
    {
        Clovek c = (Clovek) i.next();
        c.vypisInfo();
    }
}
```

Seznamy – příklad (1.4.2)

Vyvolá výjimku `java.lang.ClassCastException`

```
public static void main(String[] args) {
    List seznam = new ArrayList();
    seznam.add(new Clovek("Ferda"));
    seznam.add(new Clovek("Franta"));
    seznam.add(new Clovek("Ferenc"));

    for (Iterator i = seznam.iterator(); i.hasNext();)
    {
        String c = (String) i.next();
    }
}
```

Seznamy – příklad (5.0)

Při překladu předchozího kódu:

```
Note: ListDemo2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

```
javac -Xlint:unchecked ListDemo2.java

ListDemo2.java:9: warning: [unchecked] unchecked call
      to add(E) as a member of the raw type java.util.List
  seznam.add(new Clovek("Ferda"));
...
```

Seznamy – příklad (5.0)

Vytvoříme seznam, naplníme jej a chodíme po položkách iterátorem.

```
public static void main(String[] args) {
  List<Clovek> seznam = new ArrayList<Clovek>();
  seznam.add(new Clovek("Ferda"));
  seznam.add(new Clovek("Franta"));
  seznam.add(new Clovek("Ferenc"));

  for (Iterator<Clovek> i = seznam.iterator(); i.hasNext();)
  {
    Clovek c = i.next();
    c.vypisInfo();
  }
}
```

Seznamy – příklad

Vytvoříme seznam, naplníme jej a chodíme po položkách speciálním iterátorem.

```
public static void main(String[] args) {
    List seznam = new ArrayList();
    seznam.add(new Clovek("Ferda"));
    seznam.add(new Clovek("Franta"));
    seznam.add(new Clovek("Ferenc"));

    ListIterator li = seznam.listIterator(1);

    Clovek c = (Clovek) li.previous();
    c.vypisInfo();

    c = (Clovek) li.next();
    c.vypisInfo();
}
```